•
•
•

PROBLEMS

8.1 Consider the following sequence of instructions

```
Add   #20,R0,R1
Mul   #3,R2,R3
And   #$3A,R2,R4
Add   R0,R2,R5
```

In all instructions, the destination operand is given last. Initially, registers R0 and R2 contain 2000 and 50, respectively. These instructions are executed in a computer that has a four-stage pipeline similar to that shown in Figure 8.2. Assume that the first instruction is fetched in clock cycle 1, and that instruction fetch requires only one clock cycle.

(*a*) Draw a diagram similar to Figure 8.2*a*. Describe the operation being performed by each pipeline stage during each of clock cycles 1 through 4.

(*b*) Give the contents of the interstage buffers, B1, B2, and B3, during clock cycles 2 to 5.

8.2 Repeat Problem 8.1 for the following program:

```
Add    #20,R0,R1
Mul    #3,R2,R3
And    #$3A,R1,R4
Add    R0,R2,R5
```

8.3 Instruction I_2 in Figure 8.6 is delayed because it depends on the results of I_1. By occupying the Decode stage, instruction I_2 blocks I_3, which, in turn, blocks I_4. Assuming that I_3 and I_4 do not depend on either I_1 or I_2 and that the register file allows two Write steps to proceed in parallel, how would you use additional storage buffers to make it possible for I_3 and I_4 to proceed earlier than in Figure 8.6? Redraw the figure, showing the new order of steps.

8.4 The delay bubble in Figure 8.6 arises because instruction I_2 is delayed in the Decode stage. As a result, instructions I_3 and I_4 are delayed even if they do not depend on either I_1 or I_2. Assume that the Decode stage allows two Decode steps to proceed in parallel. Show that the delay bubble can be completely eliminated if the register file also allows two Write steps to proceed in parallel.

8.5 Figure 8.4 shows an instruction being delayed as a result of a cache miss. Redraw this figure for the hardware organization of Figure 8.10. Assume that the instruction queue can hold up to four instructions and that the instruction fetch unit reads two instructions at a time from the cache.

8.6 A program loop ends with a conditional branch to the beginning of the loop. How would you implement this loop on a pipelined computer that uses delayed branching with one delay slot? Under what conditions would you be able to put a useful instruction in the delay slot?

8.7 The branch instruction of the UltraSPARC II processor has an Annul bit. When set by the compiler, the instruction in the delay slot is discarded if the branch is not taken. An alternative choice is to have the instruction discarded if the branch is taken. When is each of these choices advantageous?

8.8 A computer has one delay slot. The instruction in this slot is always executed, but only on a speculative basis. If a branch does not take place, the results of that instruction are discarded. Suggest a way to implement program loops efficiently on this computer.

8.9 Rewrite the sort routine shown in Figure 2.34 for the SPARC processor. Recall that the SPARC architecture has one delay slot with an associated Annul bit and uses branch prediction. Attempt to fill the delay slots with useful instructions wherever possible.

8.10 Consider a statement of the form

IF A>B THEN action 1 ELSE action 2

Write a sequence of assembly language instructions, first using branch instructions only, then using conditional instructions such as those available on the ARM processor.

Assume a simple two-stage pipeline, and draw a diagram similar to that in Figure 8.8 to compare execution times for the two approaches.

8.11 The feed-forward path in Figure 8.7 (blue lines) allows the content of the RSLT register to be used directly in an ALU operation. The result of that operation is stored back in the RSLT register, replacing its previous contents. What type of register is needed to make such an operation possible?

Consider the two instructions

I_1:	Add	R1,R2,R3
I_2:	Shift_left	R3

Assume that before instruction I_1 is executed, R1, R2, R3, and RSLT contain the values 30, 100, 45, and 198, respectively. Draw a timing diagram for a 4-stage pipeline, showing the clock signal and the contents of the RSLT register during each cycle. Use your diagram to show that correct results will be obtained during the forwarding operation.

8.12 Write the program in Figure 2.37 for a processor in which only load and store instructions access memory. Identify all dependencies in the program and show how you would optimize it for execution on a pipelined processor.

8.13 Assume that 20 percent of the dynamic count of the instructions executed on a computer are branch instructions. Delayed branching is used, with one delay slot. Estimate the gain in performance if the compiler is able to use 85 percent of the delay slots.

8.14 A pipelined processor has two branch delay slots. An optimizing compiler can fill one of these slots 85 percent of the time and can fill the second slot only 20 percent of the time. What is the percentage improvement in performance achieved by this optimization, assuming that 20 percent of the instructions executed are branch instructions?

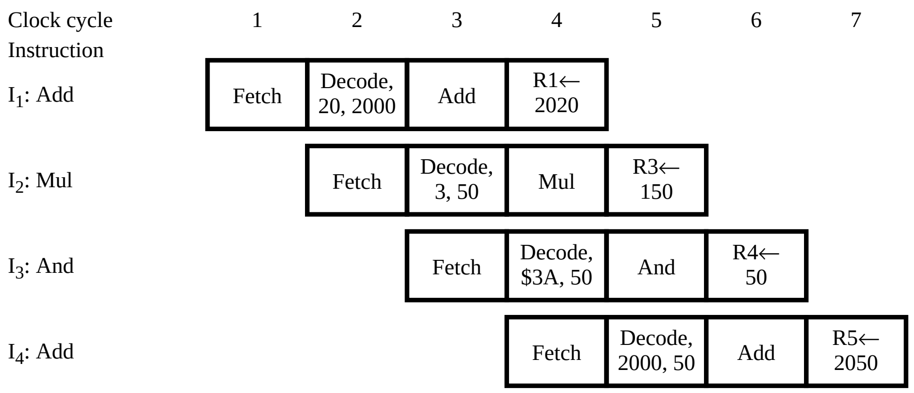
8.15 A pipelined processor uses the delayed branch technique. You are asked to recommend one of two possibilities for the design of this processor. In the first possibility, the processor has a 4-stage pipeline and one delay slot, and in the second possibility, it has a 6-stage pipeline with two delay slots. Compare the performance of these two alternatives, taking only the branch penalty into account. Assume that 20 percent of the instructions are branch instructions and that an optimizing compiler has an 80 percent success rate in filling the single delay slot. For the second alternative, the compiler is able to fill the second slot 25 percent of the time.

8.16 Consider a processor that uses the branch prediction mechanism represented in Figure 8.15*b*. The initial state is either LT or LNT, depending on information provided in the branch instruction. Discuss how the compiler should handle the branch instructions used to control "do while" and "do until" loops, and discuss the suitability of the branch prediction mechanism in each case.

8.17 Assume that the instruction queue in Figure 8.10 can hold up to six instructions. Redraw Figure 8.11 assuming that the queue is full in clock cycle 1 and that the fetch unit can read up to two instructions at a time from the cache. When will the queue become full again after instruction I_k is fetched?

Chapter 8 – Pipelining

8.1. (*a*) The operation performed in each step and the operands involved are as given in the figure below.



(*b*)

Clock cycle	2	3	4	5
Buffer B1	Add instruction (I_1)	Mul instruction (I_2)	And instruction (I_3)	Add instruction (I_4)
Buffer B2	Information from a previous instruction	Decoded I_1 Source operands: 20, 2000	Decoded I_2 Source operands: 3, 50	Decoded I_3 Source operands: \$3A, 50
Buffer B3	Information from a previous instruction	Information from a previous instruction	Result of I_1: 2020 Destination = R1	Result of I_2: 150 Destination = R3

8.2. (*a*)

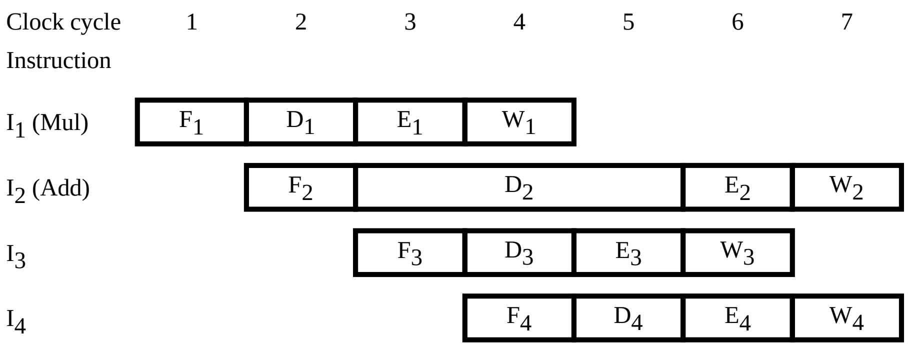
Clock cycle Instruction	1	2	3	4	5	6	7
Add	Fetch	Decode, 20, 2000	Add	R1← 2020			
Mul		Fetch	Decode, 3, 50	Mul	R3← 150		
And			Fetch	Decode, \$3A, ?	\$3A, 2020	And	R4← 32
Add				Fetch		Decode, 2000, 50	Add	R5← 2050

(*b*) Cycles 2 to 4 are the same as in P8.1, but contents of R1 are not available until cycle 5. In cycle 5, B1 and B2 have the same contents as in cycle 4. B3 contains the result of the multiply instruction.

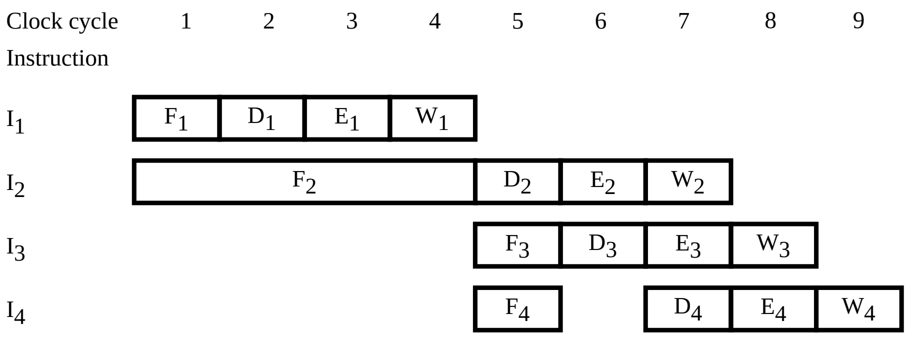
8.3. Step D_2 may be abandoned, to be repeated in cycle 5, as shown below. But, instruction I_1 must remain in buffer B1. For I_3 to proceed, buffer B1 must be capable of holding two instructions. The decode step for I_4 has to be delayed as shown, assuming that only one instruction can be decoded at a time.

Clock cycle Instruction	1	2	3	4	5	6	7	8
I_1 (Mul)	F_1	D_1	E_1	W_1				
I_2 (Add)		F_2	D_2		D_2	E_2	W_2	
I_3			F_3	D_3	E_3	W_3		
I_4				F_4		D_4	E_4	W_4

8.4. If all decode and execute stages can handle two instructions at a time, only instruction I_2 is delayed, as shown below. In this case, all buffers must be capable of holding information for two instructions. Note that completing instruction I_3 before I_2 could cause problems. See Section 8.6.1.



8.5. Execution proceeds as follows.



8.6. The instruction immediately preceding the branch should be placed after the branch.

```
LOOP   Instruction 1                     LOOP   Instruction 1
       ...                                      ...
       Instruction n − 1          ⇒             Instruction n − 1
       Instruction n                            Conditional Branch LOOP
       Conditional Branch LOOP                  Instruction n
```

This reorganization is possible only if the branch instruction does not depend on instruction n.

8.7. The UltraSPARC arrangement is advantageous when the branch instruction is at the end of the loop and it is possible to move one instruction from the body of the loop into the delay slot. The alternative arrangement is advantageous when the branch instruction is at the beginning of the loop.

8.8. The instruction executed on a speculative basis should be one that is likely to be the correct choice most often. Thus, the conditional branch should be placed at the end of the loop, with an instruction from the body of the loop moved to the delay slot if possible. Alternatively, a copy of the first instruction in the loop body can be placed in the delay slot and the branch address changed to that of the second instruction in the loop.

8.9. The first branch (BLE) has to be followed by a NOP instruction in the delay slot, because none of the instructions around it can be moved. The inner and outer loop controls can be adjusted as shown below. The first instruction in the outer loop is duplicated in the delay slot following BLE. It will be executed one more time than in the original program, changing the value left in R3. However, this should cause no difficulty provided the contents of R3 are not needed once the sort is completed. The modified program is as follows:

```
        ADD       R0,LIST,R3
        ADD       R0,N,R1
        SUB       R1,1,R1
        SUB       R1,1,R2
OUTER   LDUB      [R3+R1],R5   Get LIST(j)
        LDUB      [R3+R2],R6   Get LIST(k)
INNER   SUB       R6,R5,R0
        BLE,pt    NEXT
        SUB       R2,1,R2      k ← k−1
        STUB      R5,[R3+R2]
        STUB      R6,[R3+R1]
        OR        R0,R6,R5
NEXT    BGE,pt,a  INNER
        LDUB      [R3+R2],R6   Get LIST(k)
        SUB       R1,1,R1
        BGT,pt    OUTER
        SUB       R1,1,R2
```

8.10. Without conditional instructions:

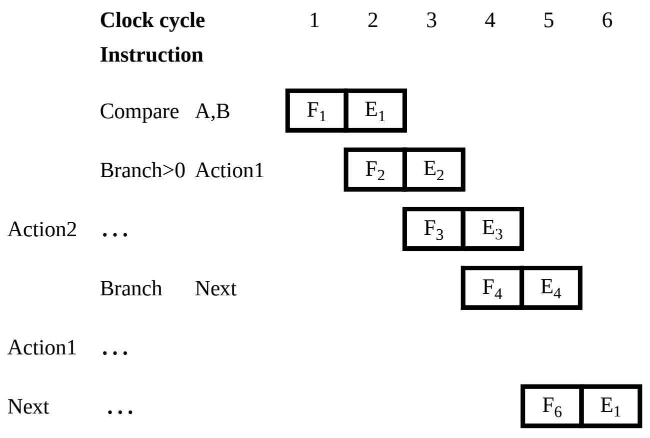
```
            Compare     A,B         Check A – B
            Branch>0    Action1
Action2     . . .       . . .       One or more instructions
            Branch      Next
Action1     . . .       . . .       One or more instructions
Next        . . .
```

If conditional instructions are available, we can use:

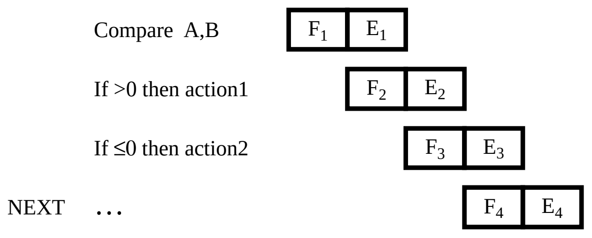
```
        Compare   A,B     Check A – B
        . . .     . . .   Action1 instruction(s), conditional
        . . .     . . .   Action2 instruction(s), conditional
Next    . . .
```

In the second case, all Action 1 and Action 2 instructions must be fetched and decoded to determine whether they are to be executed. Hence, this approach is beneficial only if each action consists of one or two instructions.

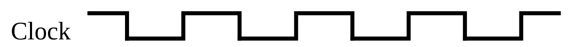
Without conditional instructions



With conditional instructions



8.11. Buffer contents will be as shown below.



Cycle No.	3	4	5
ALU Operation	+	Shift	O_3
R3	45	130	260
RSLT	198	130	260

8.12. Using Load and Store instructions, the program may be revised as follows:

```
INSERTION   Test       RHEAD
            Branch>0   HEAD
            Move       RNEWREC,RHEAD
            Return
HEAD        Load       RTEMP1,(RHEAD)
            Load       RTEMP2,(RNEWREC)
            Compare    RTEMP1,RTEMP2
            Branch>0   SEARCH
            Store      RHEAD,4(RNEWREC)
            Move       RNEWREC,RHEAD
            Return
SEARCH      Move       RHEAD,RCURRENT
LOOP        Load       RNEXT,4(RCURRENT)
            Test       RNEXT
            Branch=0   TAIL
            Load       RTEMP1,(RNEXT)
            Load       RTEMP2,(RNEWREC)
            Compare    RTEMP1,RTEMP2
            Branch<0   INSERT
            Move       RNEXT,RCURRENT
            Branch     LOOP
INSERT      Store      RNEXT,4(RNEWREC)
TAIL        Store      RNEWREC,4(RCURRENT)
            Return
```

This program contains many dependencies and branch instructions. There very few possibilities for instruction reordering. The critical part where optimization should be attempted is the loop. Given that no information is available on branch behavior or delay slots, the only optimization possible is to separate instructions that depend on each. This would reduce the probability of stalling the pipeline.

The loop may be reorganized as follows.

```
LOOP      Load       RNEXT,4(RCURRENT)
          Load       RTEMP2,(RNEWREC)
          Test       RNEXT
          Load       RTEMP1,(RNEXT)
          Branch=0   TAIL
          Compare    RTEMP1,RTEMP2
          Branch<0   INSERT
          Move       RNEXT,RCURRENT
          Branch     LOOP
INSERT    Store      RNEXT,4(RNEWREC)
TAIL      Store      RNEWREC,4(RCURRENT)
          Return
```

Note that we have assumed that the Load instruction does not affect the condition code flags.

8.13. Because of branch instructions, 120 clock cycles are needed to execute 100 program instructions when delay slots are not used. Using the delay slots will eliminate 0.85 of the idle cycles. Thus, the improvement is given by:

$$\frac{120}{120 - 20 \times 0.85} = 1.081$$

That is, instruction throughput will increase by 8.1%.

8.14. Number of cycles needed to execute 100 instructions:

Without optimization	140
With optimization $(140 - 20 \times 0.85 - 20 \times 0.2)$	127

Thus, throughput improvement is $140/127 = 1.102$, or 10.2%

8.15. Throughput improvement due to pipelining is n, where n is the number of pipeline stages.

	Number of cycles needed to execute one instruction:	Throughput
4-stage:	$1.20 - 0.8 \times 0.20 = 1.04$	$4/1.04 = 3.85$
6-stage:	$1.4 - 0.8 \times 0.2 - 0.25 \times 0.2 = 1.19$	$6/1.19 = 5.04$

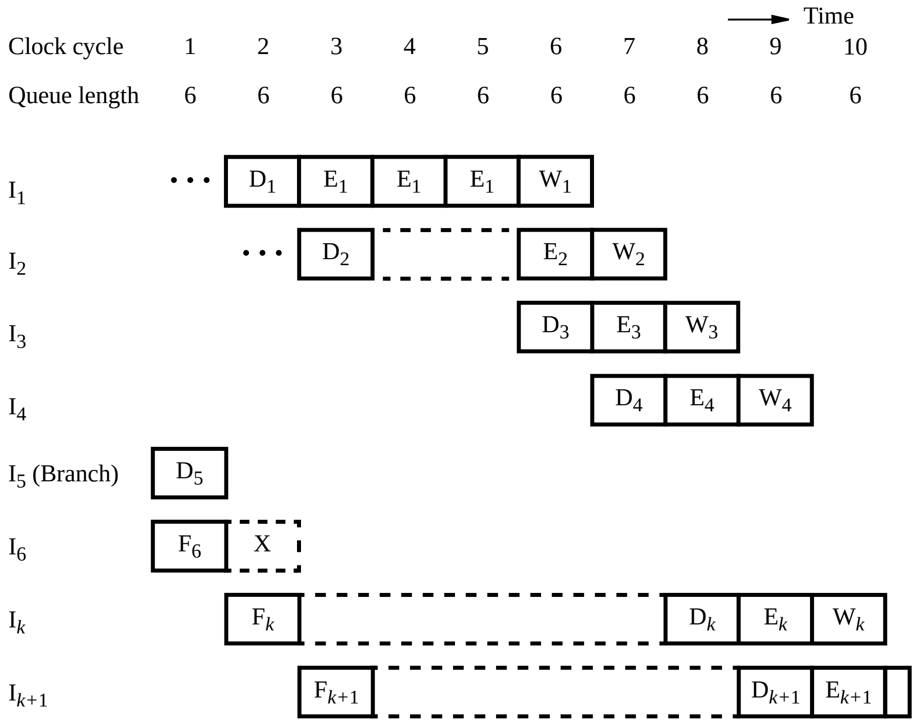
Thus, the 6-stage pipeline leads to higher performance.

8.16. For a “do while” loop, the termination condition is tested at the beginning of the loop. A conditional branch at that location will be taken when exiting the loop. Hence, it should be predicted not taken. That is, the state machine should be started in the state LNT, unless the loop is not likely to be executed at all.

A “do until” loop is executed at least once, and the branch condition is tested at the end of the loop. Assuming that the loop is likely to be executed several times, the branch should be predicted taken. That is, the state machine should be started in state LT.

8.17. An instruction fetched in cycle j reaches the head of the queue and enters the decode stage in cycle $j + 6$. Assume that the instruction preceding I_1 is decoded and instruction I_6 is fetched in cycle 1. This leads to instructions I_1 to I_6 being in the queue at the beginning of cycle 2. Execution would then proceed as shown below.

Note that the queue is always full, because at most one instruction is dispatched and up to two instructions are fetched in any given cycle. Under these conditions, the queue length would drop below 6 only in the case of a cache miss.


Time
Clock cycle 1 2 3 4 5 6 7 8 9 10
Queue length 6 6 6 6 6 6 6 6 6 6
I_1 D_1 E_1 E_1 E_1 W_1
I_2 D_2 E_2 W_2
I_3 D_3 E_3 W_3
I_4 D_4 E_4 W_4
I_5 (Branch) D_5
I_6 F_6 X
I_k F_k D_k E_k W_k
I_{k+1} F_{k+1} D_{k+1} E_{k+1}